Smart cards and the OpenCard Framework

Learn how to implement a card terminal and use a standard API for interfacing to smart cards from your browser

Summary
The OpenCard Framework provides programmers with an interface for the development of smart card applications in Java. Implementations of OpenCard can be 100% pure Java, or they can use existing card terminal implementations (a.k.a. *smart-card readers*) such as PC/SC. OpenCard differs from PC/SC in that it promises to provide a uniform application interface for building smart card applications on the emerging new platforms, such as network computers, phones, automatic teller machines, and cable TV boxes.

This article, the second in a series on smart cards, describes in detail the design objectives of the card terminal part of OpenCard. We then test the design objectives by interfacing two smart card readers to OpenCard. We provide two fully worked out examples of implementations using Java with some C code thrown in to access the serial ports. We implement an OpenCard card terminal for two readers: the IBM 5948 card terminal, and the Reflex20 PCCARD reader from Schlumberger. Finally, we'll discuss how to use OpenCard from a browser. *(4,800 words)*

By Rinaldo Di Giorgio

With special contribution by Peter Trommler

The previous **Java Developer** column, "Smart cards: A primer", gave a general overview of smart cards and how they work. It included a section on smart card standards, introducing the concept of OpenCard. As described in the first article, OpenCard is an open standard that provides interoperability of smart card applications across NCs, POS terminals, desktops, laptops, set tops, and PDAs. OpenCard can provide 100% pure Java smart card applications. Smart card applications often are not pure because they communicate with an external device or use libraries on the client. In this article we will provide two implementations to two different card readers, demonstrating how you would add support for card readers to OpenCard. We are hopeful that ports for Litronic, Gemplus, Schlumberger, Bull, Toshiba, and SCM will be available soon, compliments of OpenCard and *JavaWorld*.

Introduction

In order to use a smart card, you need to be able to read the card and communicate with it using an application. OpenCard provides a framework for this by defining interfaces that must be implemented. The OpenCard framework defines several of these interfaces. Once these interfaces are implemented, you can use other services in the upper layers of the API. For example, with a properly interfaced reader, OpenCard can start a Java card agent whenever the card is inserted. The card agent can then communicate with applications on the smart card via the card terminal in the context of a session.

This article will teach you how to interface card terminals to OpenCard. Future articles will discuss how to write an agent. A small test application, which gets the ATR (Answer to Reset) string is provided. The ATR is fundamental to smart cards. We will take the OpenCard development kit and explain implementations for two different smart card

readers using the Card Terminal Interface. The techniques discussed in the article for powering up readers, starting card sessions, and the use of Protocol Data Units and Application Protocol Data Units can be reused for most of the readers on the market.

While it's not necessary to use OpenCard in creating 100% pure Java smart card applications, without it developers are forced to use home-grown interfaces to smart cards. (For a detailed explanation of what 100% pure really means see the Resources section.) OpenCard also provides developers with an interface to PC/SC (a smart card applicatior interface developed by Microsoft and others for communicating with smart cards from Win32-based platforms for PCs) for use of existing devices on Win32 platforms. Read on and learn how to use smart cards with your browser.

OpenCard architecture: An overview

OpenCard provides an architecture for developing applications in Java that utilize smart cards or other ISO 7816-compliant devices on different target platforms such as Windows, network computers, Unix workstations, Webtops, set tops, and so on. The OpenCard Framework provides an application programming interface (API), which allows you to register cards, look for cards in readers, and optionally have Java agents start up when cards are inserted in the reader. The architecture of OpenCard is depicted in Figure 1.

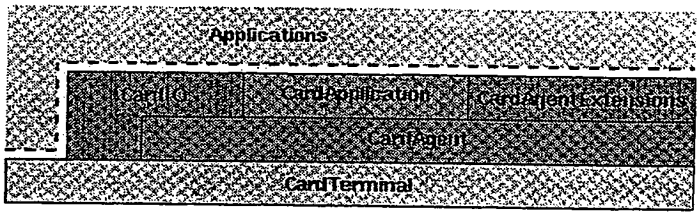

Figure 1. OpenCard Framework architecture

The architecture of the OpenCard Framework is made up of the `CardTerminal`, the `CardAgent`, the Agents and/or applications that interact with these components. OpenCard consists of four Java packages with the prefix *opencard*:

1. *application*
2. *io*
3. *agent*
4. *terminal*

The terminal package in OpenCard

The packages *opencard.application* and *opencard.io* provide the high-level API used by the application developer. The services needed by the high-level API are carried out by classes in the *opencard.agent* and *opencard.terminal* packages. The *opencard.agent* package abstracts the functionality of the smart card through the `CardAgent`. Package *opencard.terminal* abstracts the card terminals (also known as *card readers*). Understanding the structure of the *opencard.terminal* package is required to understand the sample implementations of card terminals provided in this article.

A card terminal abstracts the device that is used in a computer system to communicate with a smart card. The *opencard.terminal* package contains classes to represent the card-terminal hardware, to interact with the user, and to manage card-terminal resources. Not all readers have these abilities. When implementing a reader that doesn't have keyboard entry, we will use the `UserInteractionHandler`.

Card terminal representation

Each card terminal is represented by an instance of class `CardTerminal` that defines the abstract OpenCard-compliant card terminal. A card terminal may have one or more slots for smart cards and optionally a display and a keyboard or PIN pad. The slots of a card terminal are represented by instances of the abstract class `Slot`, which offers methods to wait for a card to be inserted, to communicate with the card, and to eject it (if possible).

User interaction

Using a smart card requires interaction with the user -- for card-holder verification. The interface `UserInteraction` provides for this functionality. It provides methods to write a message onto the display and receive input from the user. Card terminals that do not support all user interaction features can make use of the `UserInteractionHandler`, which implements a `UserInteraction` as a graphical user interface based on the abstract windowing toolkit (AWT).

Resource management

Cards and card readers require resource management so that agents can be granted the level of access control they require. Resource management provides for the sharing of card terminals and the cards inserted in them among the agents in the system. For example, say you are using your smart card to sign a document at the same time that a high-priority mail message comes in that needs to be decoded using your smart card. Resource management arbitrates the access to the `CardTerminal` and the correct port.

The resource management for card terminals is achieved by the `CardTerminalRegistry` class of OpenCard. There is only one instance of `CardTerminalRegistry`: the system-wide card terminal registry. The system-wide card terminal registry keeps track of the card terminals installed in the system. The card terminal registry can be configured from properties upon system start up or dynamically through `register` and `unregister` methods to dynamically add or remove card terminals from the registry.

During the registration of a card terminal, a `CardTerminalFactory` is needed to create an instance of the corresponding implementation class for the card terminal. The card terminal factory uses the type name and the connector type of the card terminal to determine the `CardTerminal` class to create. The concept of a card terminal factory allows a card terminal manufacturer to define a mapping between user-friendly type names and the class name.

Sample implementation: IBM card terminal

In this section, we'll describe the integration of the IBM 5948 card terminal into OpenCard. The IBM 5948 card terminal has one slot for smart cards, an LCD display, and a PIN pad. It is connected to the workstation or PC via a serial port. More information on this reader is available in the Resources section.

In order to access a card terminal from within OpenCard, an implementation for both abstract classes `CardTerminal` and `Slot` must be provided. These have been named `IBM5948CardTerminal` and `IBM5948Slot`, respectively. In addition, an appropriate `CardTerminalFactory` named `IBMCardTerminalFactory` is needed. The terminal implementation consists of package *com.ibm.zurich.smartcard.terminal.ibm5948*. Figure 2 depicts the inheritance relationships between the classes of *opencard.terminal*, the Java classes, and the terminal implementation. The class diagram also contains class `IBM5948Driver`, which does not implement any abstract class of OpenCard but serves as a Java interface to the terminal driver library written in C.

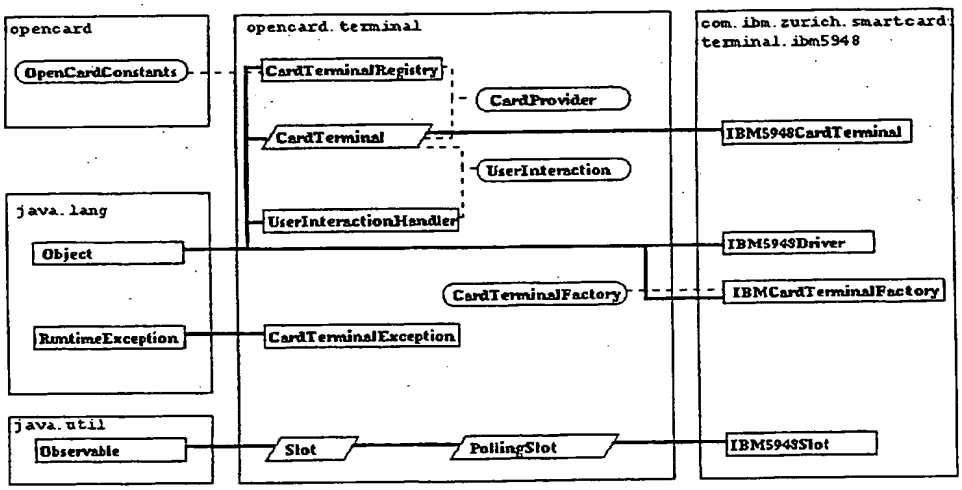

Figure 2. Inheritance diagram for the IBM 5948 card terminal

We assume that the terminal is already connected to the workstation or PC, and that the serial port is configured to work with the terminal. In the following section, we describe the design and implementation of the driver, the terminal, the slot, and the card terminal factory. The configuration of the card terminal registry also is provided.

The card terminal driver

The card terminal is shipped with a driver that is available as a dynamic link library (DLL). The DLL has a C API that offers the functions `CT_init`, `CT_data`, and `CT_close`:

- The function `CT_init` is used to open a connection to a card terminal that is connected to a certain serial port. After the connection has been established, protocol data units (PDU) can be exchanged with the card terminal and APUs can be exchanged with the smart card that is plugged into the slot of the terminal via the `CT_data` function.
- The `CT_data` call is used to send one PDU and retrieve the response from the terminal or the smart card, respectively.
- The `CT_close` function is used to close the connection to the card terminal and release any resources.

Success or failure of all three API calls is indicated by the return code.

The Java API

Similar to the C API, we define a Java API for the card terminal driver. The Java API for the card terminal consists of class `IBM5948Driver`, which has native methods calling the C API. We decided to implement as much functionality as possible in Java and have only some "glue" code written in C. In fact, the parameters of the `ctInit` and `ctClose` method are just passed on to the respective C API function. Since arrays are organized differently in C and Java, they need to be handled by calls to the Java Native Interface (JNI) API of the virtual machine. The native methods return the return code of the C API. The implementation of the `ctData` method is shown below:

```
JNIEXPORT jint JNICALL
Java_com_ibm_zurich_smartcard_terminal_ibm5948_IBM5948Driver_ctData(JNIEnv
*env,
                                                                    jobject that,
                                                                    jbyte destination,

                                                                    jbyteArray
command,
                                                                    jint
commandLength,
                                                                    jbyteArray
response,
                                                                    jint responseMax)
{
  short rc;
  unsigned char sad = HOST;
  unsigned char dad = destination;
  unsigned short responseLength = (unsigned short)responseMax;
  unsigned char *commandArray;
  unsigned char *responseArray;
  jclass cls = (*env)->GetObjectClass(env, that);
  jfieldID fid;
  jint ctn;

  fid = (*env)->GetFieldID(env, cls, "ctNumber", "I");
  if(fid == NULL) {
    return(CT_ERR_HTSI);
  }
  ctn = (*env)->GetIntField(env, that, fid);

  commandArray = (unsigned char *) (*env)->GetByteArrayElements(env, command,
                                                                0);
  responseArray = (unsigned char *) (*env)->GetByteArrayElements(env, response,
                                                                 0);

  rc = CT_DATA(ctn, &dad, &sad,
               commandLength, commandArray,
               &responseLength, responseArray);

  (*env)->ReleaseByteArrayElements(env, command, (signed char *)commandArray,
                                   0);
  (*env)->ReleaseByteArrayElements(env, response, (signed char *)responseArray,
                                   0);
```

```
    fid = (*env)->GetFieldID(env, cls, "responseLength", "I");
    if(fid == NULL) {
      return(CT_ERR_HTSI);
    }

    (*env)->SetIntField(env, that, fid, responseLength);
    return rc;
}
```

The native methods described above mimic the C API in Java. The reason for this was to have as little C code to maintain as possible. On top of the native methods, which are private, the methods `init`, `data`, and `close` are implemented. They call the native methods and throw an exception if the return code indicates an error. In the case of the data method, the response byte array is returned upon a successful completion of the native method call. The example below shows the data method:

```
synchronized byte[] data(byte destination, byte[] pdu) throws CardTerminalException {
  int rc = ctData(destination, pdu, pdu.length, response, response.length);

  if (rc == CT_OK) {
    byte[] result = new byte[responseLength];
    System.arraycopy(response, 0, result, 0, responseLength);
    return result;
  }
  else
    throw new CardTerminalException(rc2String(rc));
}
```

In order to keep memory management inside Java, a buffer response for the answer from the terminal is allocated once and passed on to the native code. Since the C API is not re-entrant, the methods of `IBM5948Driver` must be declared synchronized.

Implementing the card terminal

The card terminal is controlled by submitting control PDUs to the data method of the `IBM5948Driver`. The format of the control PDUs is ISO 7816-4 compliant. This allows us to deploy class `opencard.agent.CommandPDU` to construct the PDUs and `opencard.agent.ResponsePDU` to handle the responses.

The `IBM5948CardTerminal` class extends class `CardTerminal`. The constructor initializes the super class and instantiates the driver. Then it instantiates the array to hold the slots, and instantiates one instance of `IBM5948Slot` to represent the only slot of the IBM 5948 card terminal.

The abstract methods of `CardTerminal` are implemented using the command set of the IBM 5948 card terminal. Before constructing a command PDU, the actual parameters of the method are checked against the features of the card terminal. For the `UserInteraction`, for example, we must use the `UserInteractionHandler` if we want to read alpha-numeric input, since our terminal has only a pinpad and no keyboard.

If the terminal supports the request, a command PDU is constructed and passed on to the driver's data method. The response PDU is then checked for errors, and, in the case of an error, an appropriate `CardTerminalException` is thrown. The interaction with the terminal driver looks like this:

```
byte[] text;
// ... put the message into text
CommandPDU cpdu = new CommandPDU(TERMINAL, DISPLAY, (byte)0, (byte)0, text);
ResponsePDU rpdu = driver.data(CARDTERMINAL, cpdu.toTlByteArray());
```

The implementation of the `waitForCard` method is straightforward since only one slot exists. With only one slot in the terminal, waiting for a card in the terminal is equivalent to waiting at the slot itself. Hence, we call the `waitForCard` method of the slot instance.

Implementing the Slot

A slot as defined in OpenCard can either extend the `Slot` class or the `PollingSlot` class, depending on whether or not the insertion and removal of the smart card causes the virtual machine to generate an event. Normally, such an event is not generated, so a thread must be started to poll for smart card presence in the slot. This is carried out by the `PollingSlot` class. Our driver is implemented in C, and, as it does not generate the event, we extend

`PollingSlot`.

The IBM 5948 card terminal offers three slot-related control commands: "get status," "request card," and "eject card":

- The "get status" command returns information on whether a smart card has been inserted, the power status of the card, the status of the terminal's LEDs, and the configuration of the card terminal itself.
- The "request card" command instructs the terminal to wait for the user to insert a card within a given time interval. Once the card is inserted, it is powered up, reset, and the answer to reset (ATR) is returned.
- The "eject card" command does not really eject the card but powers down the card and waits for the user to remove it.

The `cardPresent` method is implemented by the "get status" command. The methods `powerUpCard` and `waitForCardID` must be implemented through the "request card" command. The difference between the two methods is that `waitForCardID` returns a `CardID` object and `powerUpCard` doesn't.

The IBMCardTerminalFactory

The implementation of the `CardTerminalFactory` is shown below. As long as only one terminal type is supported, all you need to do is check for the type parameter and instantiate the `IBM5948CardTerminal` class.

The `IBMCardTerminalFactory` uses the factory design pattern, which returns a specific implementation for the specified type of reader. For example, if there were 20 readers from IBM, there would be 20 possible types of readers returned from this factory class.

```
package com.ibm.zurich.smartcard.terminal.ibm5948;

import opencard.terminal.*;

public class IBMCardTerminalFactory implements CardTerminalFactory {

  public CardTerminal createCardTerminal(String name, String type,
                                         String connector)
    throws ClassNotFoundException, CardTerminalException {
      IBM5948CardTerminal terminal = null;

      if (type.equals("IBM5948-B02")) {
        terminal = new IBM5948CardTerminal(name, type, connector);
      }
      else {
        throw new ClassNotFoundException("Type unknown: " + type);
      }
      return terminal;
  }
}
```

Configuration of the CardTerminalRegistry

The new terminal must be registered with the OpenCard card terminal registry before it can be accessed. For a test program, it is recommended that you register the new card terminal dynamically using the `register` method:

```
CardTerminalRegistry registry = CardTerminalRegistry.registry();
registry.register("Zueri",
                  "com.ibm.zurich.smartcard.terminal.ibm5948.IBMCardTerminalFactory",
                  "IBM5948-B02",
                  "1");
```

Once the card terminal has been debugged, it can be registered automatically from the OpenCard properties file called *opencard.properties*. Assuming the new card terminal is the first OpenCard card terminal, the following lines must be added:

```
# ... the "Zueri" terminal
OpenCard.CardTerminal.0.name=Zueri
OpenCard.CardTerminal.0.factory=com.ibm.zurich.smartcard.terminal.ibm5948.IBMCardTermi
```

```
OpenCard.CardTerminal.0.type=IBM5948-B02
OpenCard.CardTerminal.0.address=1}
```

Building the example for the IBM driver

To build the IBM 5948 files, you can get the release from the OpenCard Web site in the Resources section and use GNUmake (make utility from the GNU group). This release is for more experienced programmers because it involves the entire source hierarchy. Later in this article, we will offer a minimal version of the release that keeps all the functionality in .jar files.

Supporting a Reflex20 PCCard reader in the OpenCard Framework

In this section, we will expand on all of the little pieces required to implement the concrete classes `CardTerminal`, `Slot`, and `CardTerminalFactory` for another type of reader, the Reflex20 from Schlumberger. (For a link to information on this reader, see the Resources section.) One of the values of a standard is that once you understand it, you are supposed to be able to repeat it without having to go through a significant learning curve each time. With the knowledge we gained from the previous section, we should be able to quickly write the interface code for the Reflex20 driver. One of the major advantages of Java is that you can support new functionality quickly; it is easier to program with Java, especially in the area of device drivers. Classically, device drivers have been located in the kernel or the operating system and often have real-time performance constraints. For smart cards, this is not the case because the driver is not running as part of the VM and smart cards are very slow. So if some new card terminal comes along, you should be able to interface to it quickly and have your old applications run without modification. Of course, this isn't always true if there is new functionality in the reader -- but at least we are not taking a step backward.

Background material on JNI and PCCard support for OpenCard

The terminal implementation consists of package *ora.smartcard.terminal.reflex20*. The Reflex20 is a PCCARD reader. This type of reader plugs into a PCCard socket (also referred to as a PCMCIA socket). Most laptops come with one or two PCCARD slots. PCCARD readers are available as add-on cards for PCs or as external devices. You can look into PCCard support for your computer in the Resources section. In order to write drivers, you need hardware, so get yourself a Reflex20 reader or another reader of choice and some smart cards.

The Reflex20 card terminal driver

The Reflex20 reader comes with a DLL and a .lib file that implement the C API. We proceed as we did earlier and use the Java native bindings to call these methods. A useful tip for looking at DLLs is provided with the MSVC and is standard equipment on Unix platforms. This DLL offers the functions *CT_init*, *CT_data*, and *CT_close*. I have included two dumps of the two relevant DLLs, the first being the DLL that calls the methods in the second DLL. I have included the dumps because readers have asked how to debug library loading errors. I always start with the signatures and make sure they are correct and match what the Java VM expects them to be.

The first dumpbin is for the DLL, which calls the `CT_init` methods and implements the native methods for `Reflex20Drvier.java`.

```
Microsoft (R) COFF Binary File Dumper Version 5.00.7022

Copyright (C) Microsoft Corp 1992-1997. All rights reserved.

Dump of file Reflex20Driver_w32.dll

File Type: DLL

Section contains the following Exports for Reflex20Driver_w32.dll

0 characteristics

3477B204 time date stamp Sun Nov 23 04:33:08 1997

0.00 version

1 ordinal base

3 number of functions

3 number of names
```

```
ordinal hint name

1 0 _Java_com_ora_smartcard_terminal_reflex20_Reflex20Driver_ctClose@12
(0000119A)

2 1 _Java_com_ora_smartcard_terminal_reflex20_Reflex20Driver_ctData@28
(0000101F)

3 2 _Java_com_ora_smartcard_terminal_reflex20_Reflex20Driver_ctInit@16
(00001000)

Summary

4000 .data

1000 .idata

1000 .rdata

1000 .reloc

4000 .text
```

What is interesting about this dump is the signature methods generated for the C functions. It is very important that the signatures match or you will get runtime errors. As mentioned earlier in the Java API section, JNI is needed to interface to native method libraries. The JNI convention is really very logical. Always prepend *_Java*, then provide the fully qualified name of your package, followed by the name of the method. The "@" stuff is appended by Windows.

The second file, below -- Ctscrw95.dll -- contains the reader-specific interface functions; it simply contains the implementations of the `CT_init`, `Ct_data`, `CT_close` methods that are called from the DLL above. Consult the Resources section to review JNI.

```
Microsoft (R) COFF Binary File Dumper Version 5.00.7022

Copyright (C) Microsoft Corp 1992-1997. All rights reserved.

Dump of file native\Ctscrw95.dll

File Type: DLL

Section contains the following Exports for CTSCRW95.dll

0 characteristics

323D2FFF time date stamp Mon Sep 16 10:46:23 1996

0.00 version

1 ordinal base

5 number of functions

5 number of names

ordinal hint name

3 0 CT_close (00002F10)

4 1 CT_data (00002FE0)

2 2 CT_init (00002E00)

5 3 Get_VersionString (00004490)

1 4 WEP (00002D80)
```

```
Summary

3000 .data

1000 .idata

1000 .rdata

1000 .reloc

6000 .text
```

The CT API type of interface shown above is common on many readers listed in the Resources section. Because the OpenCard native interface has been modeled on CT_init, CT_data and CT_close API, it will be easy to interface to the card terminal. If you need to review native method interfaces, take a look at the articles in the Resources section that discuss this topic. The file *Reflex20Driver.c* is almost identical to the *IBM5948Driver.c* file. Some of the names were changed to make it specific to the Reflex20, and of course, to get the correct signature so that the Java VM can find these native methods at runtime.

Implementing the card terminal
The `Reflex20CardTerminal` class extends class `CardTerminal` and is provided below with comments. The card terminal is controlled by submitting control PDUs to the data method of the `Reflex20CardTerminal`. The format of the control PDUs is ISO 7816-4 compliant. See the Resources section for more information on this. All of the processing is the same as described in the CardTerminal section for the IBM5948 above.

The abstract methods (meaning implementation deferred for implementation times like this) of CardTerminal are:

```
public abstract class CardTerminal implements CardProvider, UserInteraction
{

protected abstract Properties internalFeatures(Properties features);

public abstract String promptUser(String prompt, CardTerminalIOControl
ioControl);

public abstract Object command(String appSpecCmd, Object appSpecParameter,
int timeout)
```

These are implemented in Reflex20CardTerminal.java, which is provided in the Resources section.

The implementation of the `waitForCard` method is usually different for each PCCard because many manufacturers use a different PDU to get information from the card terminal.

Implementing the Slot As before, `Slot` in OpenCard can extend the `Slot` class or the `PollingSlot` class, depending on whether or not the insertion and removal of the smart card can be configured to generate an event to the virtual machine. The Java code for the *Slot* file is provided in the Resources section.

The ORACardTerminalFactory
As this `ORACardTerminalFactory` will be used in future articles, we will want to have several types of `CardTerminals` available for creation by name. These names correspond to the definitions in the property file. The property file can be searched for using the usual paths, and the implementation is very similar to the `IBMCardTerminalFactory` described above. To identify devices to OpenCard, you need to specify them in the *opencard.properties* file.

The attribute value pairs described below are consulted by OpenCard for registration information. Many of these properties are documented in the OpenCard documentation. See this documentation in the Resources section.

```
JAVA_INSTALL_DIR/lib/opencard.properties

$HOME/.opencard-properties

./opencard.properties

################################################
```

```
# Card Terminal Section #

################################################

OpenCard.CardTerminal.0.name=Reflex20

OpenCard.CardTerminal.0.factory=com.ora.smartcard.terminal.reflex20.ORACardTerminalFact


OpenCard.CardTerminal.0.type=Reflex20

OpenCard.CardTerminal.0.address=0

OpenCard.terminal.trace=true

com.ora.smartcard.terminal.trace=true
```

The two trace lines turn on tracing. OpenCard has some finely instrumented code with diagnostics that can be turned on dynamically. This property file is required for every platform that supports OpenCard. The property file also can be extended so that reader- and/or card-specific information is available. The latter is not recommended.

Building the examples
While developing the Reflex20 PCCard driver, we created a very easy development environment for you to develop CardTerminal implementations. We do this by providing an *opencard.jar* file that you compile with and use on your CLASSPATH while the applet or application is running. The list of files and the DLL required to implement the Reflex20 are included below. In addition to the files listed, you need to use the *opencard.jar* file, which contains all the `opencard.jar.*` classes for your import statements.

File Purpose

Makefile Makes DLLs, jars for any of a number of makefiles, nmake, GNUmake, unix make

Runprogram Runs the GetCardID application with proper settings for PATH and CLASSPATH

Reflex20CardTerminal.java Implementation of CardTerminal for the Reflex20

ORACardTerminalFactory.java Factory that returns the requested driver using a name.

GetCardID.java Simple Program to demonstrate OpenCard functionality

Reflex20Driver.java Interface to native methods

Reflex20Constants.java Convienence Class with Constants

Reflex20Slot.java Representation of a card in a Slot as an object.

Reflex20Driver_w32.c C implementations of native methods for JRI and JNI VMs

native/Ctscrw95.lib Manufacturer provided lib file

native/Ctscrw95.DLL Manufacturer provided DLL file with ct_init, ct_data and ct_close methods.

native/libReflex20Driver_w32.DLL DLL build by the makefile needed for Reflex20driver native method implementations.

Using OpenCard from Communicator

When I started writing this article, Communicator did not support the JDK 1.1 JNI, so this example was created with JRI (Communicator's version of a native method interface). A JNI version should be available soon at the OpenCard Web site. See the Resources section for information on both of these items.

We were able to use JRI to support the IBM 5948 reader, and we expect the same code also to function for the

Reflex20 PCCard reader with some name changes. Recently, Netscape announced support for JNI with Communicator 4.0.4. This is great news for the Windows market but not so good for the Unix environment, which is my preferred development environment. The 4.0.4 release of Communicator is so new that we were unable to test our code with it. We are using JRI to access the CT-API as native methods from Java. We would like to use JNI exclusively, so we would not have to support two interfaces. (We will discuss Microsoft Internet Explorer in the next section.)

To access a smart card from your browser, you need to be able to talk to the reader. As reading and writing to devices is considered a security issue, you also need to configure your browser to support it. At the current time, the three primary browsers have different security interfaces. To get the JRI version of the interface to OpenCard working on your Communicator-enabled platform, follow these steps, starting with a version of Netscape Communicator no lower than version 4.03 on Windows 95/NT:

- Upgrade to Communicator 4.03, which you can download from one of Netscape's mirror sites.
- Install JDK 1.1 Preview 2 patch http://developer.netscape.com/software/index.html?content=jdk/download.html; this contains the links to Windows 95 and NT versions and directions on how to install the patch.
- Get the Plugin SDK, which is needed for the build process: http://ftp.netscape.com/pub/sdk/plugin/windows/winsdk40.zip. This contains *javah* and includes. Unzip the file to some directory and add a macro NETSCAPEDIR to your Defs-.gmk file in makefiles/host of the OpenCard build tree.
- Running make *jri* in directory src/com/ibm/zurich/smartcard/terminal/ibm5948 will build both the JRI version and the JNI version of the Zueri card terminal.
- Add to your environment an LD_LIBRARY_PATH environment variable, containing the path where the DLLs are located.
- A test page is located in src/com/ibm/zurich/smartcard/test/IBM5948Test.html. The applet will display the terminal type in the applet frame and set the terminal's display to "Hallo..."

Running OpenCard with Internet Explorer

Interfacing to the Microsoft Internet Explorer browser requires more work due to the Windows JDirect interface. Unfortunately, Microsoft has chosen not to implement JNI, which means you cannot reuse your JNI work. Notice the difference it makes with Communicator in reducing the amount of code you must customize. JavaSoft spent some time on the development of JNI to make it complete, using some ideas from Netscape's JRI. Microsoft seems to have some technical issue with this.

In future articles, we will provide examples of interfacing to PC/SC on Microsoft platforms. As noted above, PC/SC is a standard for supporting smart cards on Windows platforms -- and that means Windows platforms *only*. (For more on PC/SC, see the previous column.) We have already run OpenCard on AIX, Solaris, Windows 95, and NC platforms. There has been interest in porting OpenCard to point-of-sale devices and PDAs.

Conclusion

Using the OpenCard Framework for smart cards you should be able to select and interface card terminals to the framework based on cost and implementation difficulty. The implementation of the IBM 5948 card terminal code demonstrates how to perform PIN functions as well as control LEDs on the reader. The interface to the Reflex20 should easily be reusable for other PCCard type devices. Soon serial support will be part of the JDK -- so a 100 percent pure Java interface to serial readers will be possible. We will show you such an implementation in the future. In next month's **Java Developer** column on smart cards, we will discuss JavaCard and provide some examples of a current implementation. After that, we will look at the 2.0 JavaCard Specification and where JavaCard is going.

To make your interfacing task easier, it is best to select a reader that has existing DLLs or .so (Unix Dynamic Link Libraries] files that are similar to the model supported by Schlumberger's Reflex20 and the IBM 5948, unless you are an extremely experienced C and Java programmer and have a solid understanding of ISO 7816. The most wonderful thing about these two readers and others is that they support the same C API, which means that it took almost no work to support the Reflex20 after understanding the IBM-supplied interface for the IBM 5948. ■

About the author

Peter Trommler is a researcher at the IBM Research Division Zurich Research Laboratory, Switzerland. He is one of the designers of the OpenCard Framework and implemented parts of the reference implementation. His research interests include smart cards, security, distributed systems, and Java.

Rinaldo S. Di Giorgio is a staff engineer for Sun Microsystems in New York City. He currently is working on the integration of many technologies into HotJava and Java, including commerce, database connectivity, portfolio management, and analytical applications for the financial and emerging genetics market. He sees Java as the technology that will minimize two great cost factors in the computer industry: distribution and code development.

Home | Mail this Story | Resources and Related Links

Resources

- You can access source code for examples in this article through a .zip or .tar file
- The Resources section from the first in the series of articles on smart cards contains a fairly exhaustive list of URLS on the smart card subject
  /javaworld/jw-12-1997/jw-12-javadev.html
- Smart cards and the Web
  http://www.netscapeworld.com/netscapeworld/nw-03-1997/nw-03-smartcard.html
- General PCCard resources
  http://beta.missilab.com/readertest/pcmcia.html
- On using native methods to get to the serial ports
  http://www.javaworld.com/javaworld/jw-07-1997/jw-07-javadev.html
- Reflex20 contact information
  http://www.pcsc.austin.et.slb.com/cyberflex/pcsc/reflex20.html
- Latest firmware and SDK for the Reflex20. If you have problems with the example, download this file and replace the DLL interface library. Follow the directions for updating the firmware.
- IBM 5948 Card Accepting Device
  http://www.chipcard.ibm.com/sc09adev.htm
- OpenCard architecture, documents, and source code for this article
  http://www.opencard.org/
- Latest version of Communicator (4.0.4) has JNI support for Windows 95 platforms
  http://developer.netscape.com/software/jdk/download.html
- 100% Pure -- the official definition. This FAQ has answers to most questions about what makes a Java applet, application, or Bean 100% pure
  http://www.javasoft.com/features/1997/may/100percent_qna.html